ALGONQUIN COLLEGE

ASSIGNMENT 3 SPECIFICATION[1] - PARSER

General View

Due Date: prior or on **April 17th, 2021 (midnight)**

- **Important Note**: Late submission can be accepted (since we are entering in the final exam week).

Earnings: 15% of your course grade

Development: Activity can be done **individually** or in teams (**only 2 students** allowed).

Purpose: **Development of a Parser, rewriting the grammar and adjusting the tables.**

- Assignment #3 consists of **two tasks**. Task 1 is related to rewriting grammar, and Task 2 involves the PLATYPUS 2.0 Parser implementation. At the end, you will be able to have a rudimentary a PLATYPUS interpreter.

The main objective is to write a ***Recursive Descent Predictive Parser*** (**RDPP**) for the **PLATYPUS *2.0*** language. You are to integrate the Parser with your existing lexical analyzer to complete the front-end of your PLATYPUS 2.0 compiler. The implementation is broken into two tasks.

Task 1: Modifying the Grammar (5 marks)

1.1. TASK OVERVIEW

To build a RDPP you need to modify the **syntactical part** of the PLATYPUS 2.0 Grammar (The Platypus Syntactic Specification). The grammar provided for you in ***PlatypusLGR_W21.pdf*** is an LR grammar (that is, a grammar suitable for LR parsing).

[1] Adapted from resources developed by Prof. Svillen Ranev (Algonquin College, 2019)

- You must transform it into an LL grammar suitable for ***Recursive Descent Predictive Parsing***. To accomplish that you should follow the steps outlined bellow.
- Check the PLATYPUS 2.0 Grammar for completeness and correctness.
- Eliminate the ***left recursion*** and apply ***left factoring*** where needed.

Note 1: Remembering RDPP

The RDPP means that you are creating a deterministic way to identify the sequence of tokens from one specific grammar. To do this, it is required to solve some problems (left recursion and left factoring) and we need to be able to detect some tokens (see the FIRST set to be used).

Some of the syntactic productions must be rewritten to make them suitable for recursive decent predictive parsing. Do not forget that our grammar (***PlatypusLGR_20F.pdf***) is an LR grammar, which must be transformed into an equivalent LL grammar.

For example, the productions of the type:

<statements> → <statement> | <statements> <statement>

can be rearranged in a convenient for transformation form

<statements> → <statements> <statement> | <statement>

Once you rearrange the production, you must eliminate the immediate left-recursion. The transformation will produce the following two new productions:

<statements> → <statement> <statementsPrime>
<statementsPrime> → <statement> <statements>

In some cases, it is possible to rework the grammar to avoid some of the transformations. This approach is often applied to production, which contain a *"left-factor"*. For example, the output statement

<output statement> → WRITE (<opt_variable list>);
| WRITE (<string literal>);

can be reworked in the following way:

<output statement> → WRITE (<output list>);
<output list> → <opt_variable list> | STR_T

where STR_T is a string literal token produced by the scanner.

The <output statement> production does not contain a left-factor anymore.

- ***TIP**: It is also possible to simplify the grammar applying formal simplification procedures (see pages 223-224 (old 183-185) in your textbook). Do not simplify the grammar for this assignment.*

- Build the FIRST set for the **syntactic** grammar.
  - After you transform the grammar you must build the **FIRST set** for each of the grammar productions (non-terminals).
  - A FIRST set for a production (nonterminal) is a set of terminals that appear as left-most symbols in any of the production alternatives.
  - When you build the FIRST set, if some of the elements of the set are non-terminals, they must be replaced by their FIRST sets and so on.
  - The final FIRST set must contain only **terminals** (tokens).
  - The elements of the set must be **unique**.
  - This is essential for the predictive parser. If they are not unique, the left factoring transformation must be applied to the corresponding production.

1.2. TASK SUBMISSION

- **Part 1 - Task 1** Submission**:**
  - Write the ***entire syntactic grammar*** and the corresponding FIRST sets.
  - Do not remove the original productions. If a production is to be transformed, write the **modification** below the original production and **indicate clearly** the type of the transformations applied to the original production.
  - Do not include the provided explanatory text. Include only the grammar productions. Write your name(s) on every page.

Tip 1: See Template

*Some examples of transforming the grammar and building the FIRST set is done for you in the document **PlatypusLGR_W21F.pdf** published on Brightspace. The transformation is indicated in blue and type of the applied transformations are indicated in red. The work you must complete is indicated with the word complete in red.*

Now you are ready for the next task. You can work on both tasks simultaneously - production by production and function by function. I strongly recommend this approach.

Task 2: Parser Implementation (10 marks)

2.1. GENERAL VIEW

In Task 1, you had changed the grammar for the PLATYPUS 3.0 programming language. Now, in Task 2, you are to create a synthetical analyzer (**parser**) for the PLATYPUS 2.0 programming language.

2.2. IMPLEMENTATION STEPS

To build the RDPP follow the steps outlined below.

Step 1

- Open also the ***parser_uncomplete.h***. Include the required system and user header files.
- TODO_01: Define one static global variable: ***lookahead*** of type ***Token***.
- TODO_02: Additionally, define a global variable **syntaxErrorNumber** of type ***int***.
- You have some **extern** variables to be used in your code. You may add additional variable declarations and constants definitions, if necessary (and it is).
- TODO_03: Create **constants** for all keywords from PLATYPUS 3.0.
- TODO_04: You have the main functions defined; however, you must define all the functions used in the parser implementation that represents nonterminal definitions from grammar. For instance: void program(void).
- So, all function prototypes, variable definitions/declarations, and constant definitions must be in ***parser.h***.

Step 2

- Open your RDPP source code file ***parser_uncomplete.c***. You can see that your code for the parser has been already created.

- Your ***startParser()*** function. You can see that the first top-down execution is already given for you: note that you are processing the main structure in PLATYPUS 2.0: PROGRAM, given by the function program(), that you need to implement later.

```
void startParser(void) {
	lookahead = processToken();
	program();
	matchToken(SEOF_T, NO_ATTR);
	printf("%s\n", "PLATY: Source file parsed");
}
```

- Your ***matchToken()*** function is also partially defined:

```
void matchToken(int tokenCode, int tokenAttribute) {
	int matchFlag = 1;
	switch (lookahead.code) {
	case KW_T:
	case REL_OP_T:
	case ART_OP_T:
	case LOG_OP_T:
		//TODO_05
	default:
		//TODO_06
	}
	if (matchFlag && lookahead.code == SEOF_T)
		//TODO_07
	if (matchFlag) {
		//TODO_08
	}
	else
		//TODO_09
}
```

EXPLANATION:

- The ***matchToken()*** is a function called all the time by parser. It tries to match the current input token (***lookahead***) and the token required by the parser, checking with the expected values.
  - The token required by the parser is represented by two integers - the token code (***tokenCode***), and the token attribute (***tokenAttribute***).
  - TODO_05: The attribute code is used only when the token code is one of the following codes: **KW_T**, **LOG_OP_T**, **ART_OP_T**, **REL_OP_T**.
  - TODO_06: In all other cases the token code is matched only.
  - TODO_07: If the match is successful and the ***lookahead*** is **SEOF_T**, the function returns.

- TODO_08: If the match is successful and the ***lookahead*** is not **SEOF_T**, the function advances to the next input token by executing the statement:

  ```
  lookahead = processToken();
  ```

  - If the new lookahead token is **ERR_T**, the function calls the error printing function ***printError()***, advances to the next input token by calling ***processToken()*** again, increments the error counter ***syntaxErrorNumber***, and returns.
  - If the match is unsuccessful, the function calls the error handler ***syncErrorHandler()*** passing ***tokenCode*** and returns.

- TODO_09: Finally, if no matching is found, you must call the ***syncErrorHandler()*** passing the Token code.

- Your ***syncErrorHandler()*** function is also partially defined:

```
void syncErrorHandler(int syncTokenCode) {
	//TODO_10
	syntaxErrorNumber++;
	while (lookahead.code != syncTokenCode) {
		//TODO_11
	}
	if (lookahead.code != SEOF_T)
	//TODO_12
}
```

EXPLANATION:

- TODO_10: The error handling function ***syncErrorHandler()*** implements a simple panic mode error recovery. The function calls ***printError()*** and increments the error counter.
- TODO_11: Then the function implements a panic mode error recovery:
  - The function advances the input token (***lookahead***) until it finds a token code matching the one required by the parser (passed to the function as ***syncTokenCode***).
  - When found, the ***lookahead*** is updated.
  - It is possible, when advancing, that the function can reach the end of the source file without finding the matching token.
  - To prevent from overrunning the input buffer, before every move the function checks if the end of the file is reached.

        - If the function looks for ***lookahead.code*** different from **SEOF_T** and reaches the end of the source file, the function calls ***exit(syntaxErrorNumber)***.
    - TODO_12: If a matching token is found and the matching token is not **SEOF_T**, the function advances the input token one more time and returns. If a matching token is found and the matching token is **SEOF_T,** the function returns.

- Your ***printError()*** function is also partially defined:

```
void printError() {
     Token t = lookahead;
     printf("PLATY: Syntax error:  Line:%3d\n", line);
     printf("*****  Token code:%3d Attribute: ", t.code);
     switch (t.code) {
          //TODO_13
     default:
          printf("PLATY: Scanner error: invalid token code: %d\n",
                  t.code);
     }
}
```

EXPLANATION:

- The error handling function ***syncErrorHandler()*** uses a switch/case structure checking for token code the appropriate output, most of time printing the attribute from the token (using the appropriate format).
- TODO_13: Implement all the cases to show the error message.
    - TIP: the matching with standard outputs – sout files – is essential to be sure that the error handler is working fine.
- The function prints the following error message:

```
PLATY: Syntax error: Line: line_number_of_the_syntax_error
       Token code: lookahead token code Attribute: token attribute
```

and returns.

For example:

```
PLATY: Syntax error:  Line: 2
-      Token code: 13 Attribute: NA
PLATY: Syntax error:  Line: 8
       Token code:  9 Attribute: 0
PLATY: Syntax error:  Line: 9
```

```
       Token code:  2 Attribute: sum
PLATY: Syntax error:  Line: 11
       Token code:  4 Attribute: 0.5
PLATY: Syntax error:  Line: 17
       Token code:  6 Attribute: Result:
PLATY: Syntax error:  Line: 21
       Token code: 16 Attribute: ELSE
```

- If the offending token is a keyword, variable identifier, or string literal you **must** use the corresponding token attribute to access and print the lexeme (keyword name, variable name, or string).
- For example, to print the keyword lexeme you must use the ***keywordTable*** defined in ***table.h***.
- **Important note:** You are not allowed to copy the keyword table in *parser.h* or *parser.c*. You must use a proper declaration to create an external link to the one defined in *table.h*.
- Similarly, you must use the string literal table to print the sting literals.

Step 3

- TODO_14: For each of your grammar productions write a function named after the name of the production. For example:

```
void program(void) {
     matchToken(KW_T, MAIN);
     matchToken(LBR_T, NO_ATTR);
     optionalStatements();
     matchToken(RBR_T, NO_ATTR);
     printf("%s\n", "PLATY: Program parsed");
}
```

Writing a production function, follow the sub steps below.

Step 3.1:

- To implement the Parser, you **must use** the modified grammar (see Task 1).
- Before writing a function, carefully analyze the production.
- If the production consists of a single production rule (no alternatives), write the corresponding function without using the **FIRST** set (see above).
- If you use the ***lookahead*** to verify in advance whether to proceed with the production and call the ***printError()*** function.

Example: The production:

<input statement> -> INPUT (<variable list>);

Should be implemented as follows:

```
void inputStatement(void) {
     matchToken(KW_T, READ);
     matchToken(LPR_T, NO_ATTR);
     variableList();
     matchToken(RPR_T, NO_ATTR);
     matchToken(EOS_T, NO_ATTR);
}
```

AND NOT like this:

```
void input_statement(void){
     if(lookahead.code == KW_T
        && lookahead.attribute.get_int== READ) {
     matchToken (KW_T, READ);
     matchToken (LPR_T, NO_ATTR);
     variableList();
     matchToken (RPR_T, NO_ATTR);
     matchToken (EOS_T, NO_ATTR);
     printf("PLATY: Input statement parsed");
     } else
        printError();
}
```

This implementation will “catch” the syntax error but will prevent the ***match()*** function from calling the error handler at the right place.

Step 3.2:

- If a production has more than one alternative on the right side (even if one of them is empty), you must use the FIRST set for the production.
- For example, the FIRST set for the <opt_statements> production is: {**KW_T**(IF), **KW_T**(WHILE) , **KW_T**(READ), **KW_T**(WRITE), **AVID_T, SVID_T**, and .
- Here is an example how the FIRST set is used to write a function for a production:

```
void optionalStatements(void) {
     switch (lookahead.code) {
     case AVID_T:
```

```
	case SVID_T:
		statements();
		break;
	case KW_T:
		if (lookahead.attribute.get_int == IF
			|| lookahead.attribute.get_int == WHILE
			|| lookahead.attribute.get_int == READ
			|| lookahead.attribute.get_int == WRITE) {
			statements();
			break;
		}
	default:
		printf("%s\n", "PLATY: Opt_statements parsed");
	}
}
```

- Pay special attention to the implementation of the empty string. If you do not have an empty string in your production, you must call the ***printError()*** function at that point.
- You are not allowed to call the error handling function ***syncErrorHandler()*** inside production functions and you are not allowed to advance the ***lookahead*** within production functions as well.
- Only ***matchToken()*** can call ***syncErrorHandler()***, and only ***matchToken()*** and ***syncErrorHandler()*** can advance ***lookahead.***
- Each function must contain a line in its header indicating the production it implements and the FIRST set for that production (see above).

Step 3.3:

- Build your parser incrementally, function by function.
- The function headers for the parser functions should contain only the following: **the grammar production the function implements**
  - **Example:**

```
/*************************************************************
 * Program statement
 * BNF: <program> -> MAIN { <opt_statements> }
 * FIRST(<program>)= {KW_T (MAIN)}.
 ************************************************************/
```

- After adding a function, test the parser thoroughly.
- Use your main program to test the parser.

Submission

3.1. What to Submit for Part 1 (Task 1 and Task 2):

- Only **digital submission** is required for this assignment.
- The submission MUST follow the Assignment Submission Standard.

Digital Submission Summary

- **Name**: Use a ZIP file including all the resources required to your assignment evaluation. Individual or teams submission must be done up to the due date.
  - The **name of the file** must be *Your Last Name* followed by the *last three digits* of your student number followed by A3.
  - For example: Name123_A3.zip.
    - If your last name is long, you can truncate it to the first 5 letters.
    - The name of the file must contain the names of both members e.g. Name123_Name456_A3.zip.
    - If you have worked in a team, you must include a **team page** also in the zip file.
- **DOCUMENTS**: The modified syntactic part of the PLATYPUS grammar and the FIRST sets must be submitted in MS Word or PDF format.
  - **Indicate** clearly all the changes to the grammar.
  - **Indicate** what kind of transformations you have used to modify the grammar.
  - NOTE: Remember that the transformed grammar must be ***equivalent*** to the original one.
- **CODE:** Compress into a **zip** file the following files:
  - Include all project's **.h** files and **.c** files.
  - Include any additional input test files if you have any.

Enjoy the assignment and do not forget that:

"*It is better to know some of the question than all of the answers*." James Thruber

Good Luck with Assignment 3!